by Jack W. Crenshaw

Roots by Iteration

Welcome to another installment of the Programmer's Toolbox. Lately, we have been looking at algorithms to find the root of a function:

$$y = f(x) \quad (1)$$

Two months ago, we looked at Newton's method, which is the first-order method of choice if you have an analytical expression for the derivative of f(x). Newton's method is fast but unreliable unless we take steps to tame it by limiting its range of search. Last month, we looked at three methods we can use if the derivative is unavailable: the method of bisection, the secant method, and the method of false position. The last two methods converge at roughly the same rate as Newton's and are called super-linear methods.

We learned that each method has its pros and cons. The bisection method is like the tortoise, slow but sure. It is simple and converges at a guaranteed pace. If the function is continuous and has a root in the region of interest, this method will find it, and we can even predict how many steps it will take to do so.

The secant method converges faster than competing methods, but like its philosophical equivalent, Newton's method, it can also oscillate or diverge. Special steps should be taken to tame the method if it is going to used in a general-purpose routine. The method of false position is much more stable and, as with the bisection method, we can guarantee convergence. Most of the time, convergence is as fast, or nearly so, as the secant method. But for some cases (and not just pathological cases either), the convergence can be much slower—even worse than the method of bisection.

Throughout this study, I've warned you to be careful applying any of these

A first-order method is based on the idea of fitting a straight line between two points lying on the function f(x).

algorithms blindly—to "know your function." Without care in implementation, all the algorithms can lead to trouble. We need a method that's more trustworthy, a method that combines the stability and trustworthiness of the method of bisection with the fast convergence of a secant or Newton's method. This month, I'm going to do even better: I'll show you a second-order method that's not only stable and trustworthy, but converges with lightning speed once we get near the root.

METHOD X

The algorithm I'm going to show you here has an interesting history. I first encountered it in the old IBM Scientific Subroutine Library, a library of Fortran functions, circa 1965. IBM described the algorithm as Mueller's method. It's based on the concept of inverse quadratic interpolation, which I'll explain more fully in a moment. Many years ago, however, I saw a description of Mueller's method and discovered that it uses direct quadratic interpolation—a critical difference, as you'll soon see. *Numerical Recipes in C* describes Brent's method, which also uses inverse quadratic interpolation and has many of the same characteristics as the IBM algorithm.[1] But the formulations, derivations, and criteria used to decide whether to accept the approximation are all different. That leaves me with a method that works very well, but no one to attribute it to. If any of you can identify the inventor of the algorithm, please let me know. Until then, I'll just call it Method X. The derivation of the acceptance criterion is my own.

A rose by any other name would smell as sweet, and Method X is sweet indeed. It shares the property of the method of bisection: If a root exists in the range specified, this algorithm will find it—guaranteed. But unlike the method of bisection, it converges extremely rapidly. If the secant method is super-linear, this one is super-quadratic and will find the root so fast it will make your head spin. Who could ask for anything more, except a name?

THE BASICS

A first-order method is based on the idea of fitting a straight line between two points lying on the function f(x). Not surprisingly, a second-order method is based on fitting a quadratic through three points.

Consider the curve of Figure 1. We can fit a quadratic through the three points shown to get the parabola described by the dashed curve. The general form of such a quadratic is:

$$y = a + bx + cx^2 \quad (2)$$

but by playing around with the coefficients, we can eliminate the first-order term to get:

$$y - y_0 = A(x - x_0)^2 \quad (3)$$

This equation describes a parabola opening up (for positive A) or down (for negative A), with vertex at the

point (x_0, y_0). By looking at the values of the function at $x = x_1$, x_2, and x_3, we could conceivably determine the constants x_0, y_0, and A. The root is then given by setting $y = 0$ to get:

$$x = x_0 \pm \sqrt{\frac{-y_0}{A}} \qquad (4)$$

This is Mueller's method. Now, looking at Equation 4, tell me: what's wrong with this picture? That's right, we have two roots (potentially complex), not just one. The function is double-valued when solved for x as a function of y. How do we decide which of the two roots is the right one? And what if the roots are complex? In that case, we get no useful solution.

While we can conceivably make tests on the results to be sure Equation 4 leads to the right root (we'll end up performing similar tests, in any case), wouldn't it be nice if our algorithm could assure us one real root in all cases?

This is where the inverse part of "inverse quadratic interpolation" comes in, and with it, the ingenuity of Method X (and Brent's method). The concept is very simple. Instead of a parabola opening along the y-axis, let's use one that opens along the x-axis; that is, it will be a quadratic in y rather than x. A function like this will always be single-valued, since each value of y gives us one and only one value of x. Since y can take on any value, we needn't worry about complex or imaginary roots. An inverse quadratic fit is shown in Figure 2.

When you're trying to fit a curve through known points, the ease of the process depends a lot on the form you choose for the function. A general form like Equation 2 may look simple and straightforward, but you'll find that when you start substituting in the values at the known points, you're in for some tedious and error-prone math. I've found a trick that often simplifies things quite a bit. Write the equation in a factored form:

$$x - x_1 = A + B(y - y_1)[1 + C(y - y_2)] \qquad (5)$$

You can see the advantage to this form when you evaluate it at the known points. At point P_1, we have:

$$x = x_1$$
$$y = y_1 \qquad (6)$$

so the equation simply gives us:

$$0 = A$$

Thus, A must be zero.

The next coefficient is equally easy

FIGURE 1
Quadratic fit.

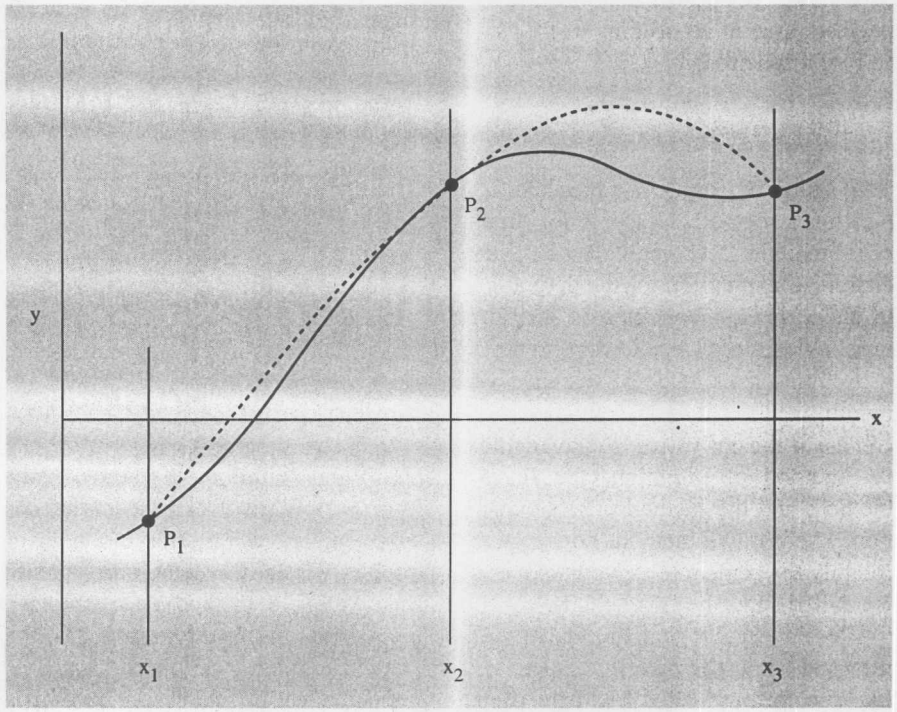
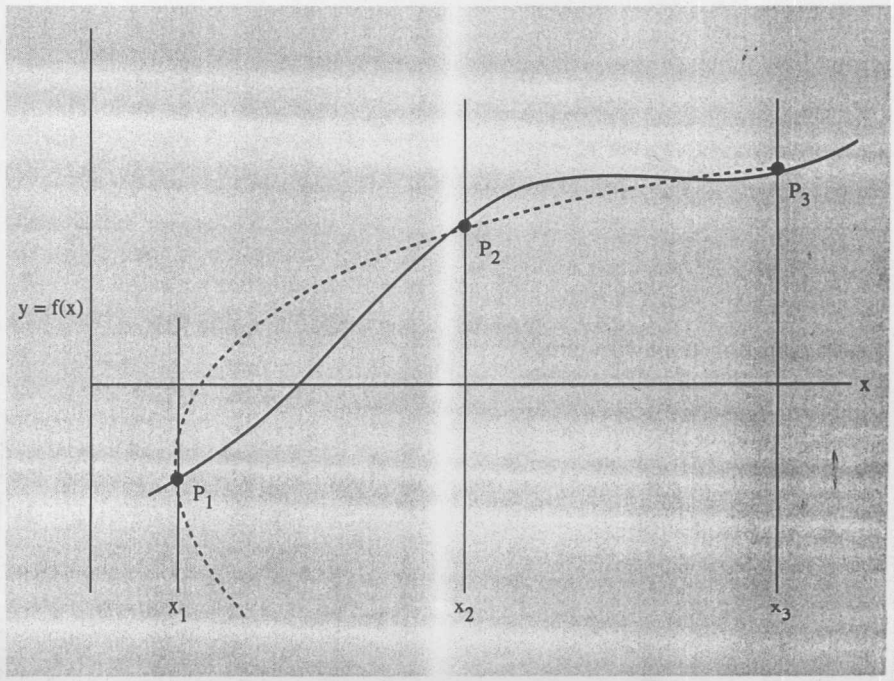


FIGURE 2
Inverse quadratic fit.



since, at point P_2, the equation becomes:

$$x_2 - x_1 = B(y_2 - y_1)$$

This gives:

$$B = \frac{x_2 - x_1}{y_2 - y_1} \tag{7}$$

As you can see, we've already determined two of the three coefficients with very little work. We get the last coefficient by evaluating at P_3:

$$x_3 - x_1 = B(y_3 - y_1)[1 + C(y_3 - y_2)]$$

Solving for C, we get:

$$1 + C(y_3 - y_2) = \frac{x_3 - x_1}{B(y_3 - y_1)}$$

Using Equation 7, this becomes:

$$1 + C(y_3 - y_2) = \frac{(x_3 - x_1)(y_2 - y_1)}{(x_2 - x_1)(y_3 - y_1)} \tag{8}$$

So far, we haven't placed any restrictions on the values of x_1, x_2, and x_3. But let's now assume that x_2 is halfway between x_1 and x_3, so:

$$x_2 - x_1 = x_3 - x_2 = h \tag{9}$$

Equation 7 becomes:

$$B = \frac{h}{y_2 - y_1} \tag{10}$$

Equation 8 becomes:

$$1 + C(y_3 - y_2) = \frac{2(y_2 - y_1)}{y_3 - y_1}$$

Continuing with the solution for C gives us:

$$C(y_3 - y_2) = \frac{2(y_2 - y_1)}{y_3 - y_1} - 1$$

$$= \frac{2(y_2 - y_1) - (y_3 - y_1)}{y_3 - y_1}$$

$$= \frac{-y_1 + 2y_2 - y_3}{y_3 - y_1}$$

and thus:

$$C = \frac{-y_1 + 2y_2 - y_3}{(y_3 - y_2)(y_3 - y_1)} \tag{11}$$

We now have all three coefficients in terms of the fitted points. Remember this "factored polynomial" trick—it's very useful, and works for polynomials of any order. As you can see, the computation of the coefficients gets progressively harder, but it will still be easier than with any other form.

Once we have the coefficients, we can find the desired root by setting y = 0 in Equation 5:

$$x = x_1 - By_1(1 - Cy_2) \tag{12}$$

At first glance, you might think we're home free at this point, since we need only solve Equation 12 for x. Unfortunately, a lot of snakes still lurk in the equations to bite the unwary. For starters, if any pair of y-values are equal, one of the terms in the denominators of Equations 7 or 11 will go to zero, and our equations blow up. I was very careful not to show the inverse quadratic fit in Figure 1. That's because $y_2 = y_3$ in that figure and no (finite) parabola opening horizontally is possible.

By requiring that points P_1 and P_3 straddle the root, we can be sure that:

$$y_1 \neq y_3$$

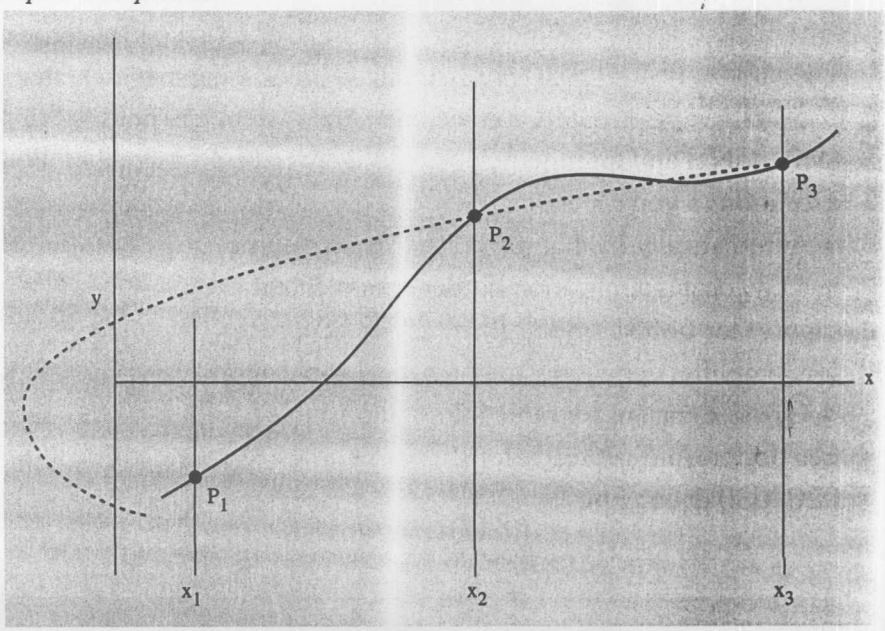
So at least one term can't vanish. But we have no control over the value of y_2, so we can't promise that one of the other two terms won't vanish. What do we do in this case? That's easy: We simply refuse to do the interpolation. In finding y_2, we have, in effect, taken one step of the bisection method. If we find that we can't do the interpolation, all is not lost. We simply take another bisection step and repeat the process. Eventually, the method must give us a useable interpolant.

Even if the terms in Equations 7 and 11 don't blow up, we might still find that our interpolation formula doesn't help. Consider the curve of Figure 3. While our formula works, it predicts a root that's well outside the range x_1 .. x_3. Since we know that the desired root is in this range, we'd be foolish to accept the interpolated value.

ACCEPTANCE CRITERION

We need a nice mathematical test to tell us when we can trust the root estimated by the inverse interpolation. The obvious approach is to simply look at the predicted value of x. If it's not

FIGURE 3
A poor interpolant.



inside the range of our search, we should refuse to accept the step. That's what's done in Brent's method. Unfortunately, we can get into trouble even then. Look at Figure 4. In this case, we get a mathematically correct fit to the data, and the predicted root is in the range of interest. But the accuracy of the fit is terrible; the predicted root, near P_3, is far from the actual one near P_1. We'd have been better off just accepting a bisection step. What's more, if we carry the method to the next step, we'll find that the next predicted root will be out of range.

Deriving a suitable acceptance criterion is tricky, and I spent a long time finding it. But the final criterion is quite simple, and easily evaluated in software. The derivation is also fascinating in its own right, so bear with me as we delve into some tricky math. The results will be worth it, I promise.

We start by assuming that the root lies between points P_1 and P_3, and neither point is the root (if it is, we're finished). This assumption means that y_1 and y_3 have different signs and so:

$$y_1 y_3 < 0$$

After bisection to get y_2, we can tell from its sign whether the root lies to its left or its right. This is the basis, of course, of the method of bisection. Naturally, we don't know which of the two cases we'll get, but I wish we did. Looking at Equation 10, we can see that the relationship between the three points is not symmetric. That is, points P_1 and P_3 enter into the equation much differently, thanks to our factored form. To avoid having to deal with multiple cases, we would like very much always to know that the root lies between P_1 and P_2. We can assure this situation by the simple expedient of relabeling the points, if necessary.

If the root lies between P_1 and P_2, y_1 and y_2 must also have opposite signs, and y_2 and y_3 must have similar ones. Thus:

$$y_1 y_2 < 0$$

$$y_2 y_3 > 0 \qquad (14)$$

Now we'll assert our acceptance criterion: the predicted root must also lie between x_1 and x_2. This is equivalent to the assertion:

$$(x_2 - x)(x - x_1) \geq 0 \qquad (15)$$

From Equation 12:

$$\begin{aligned} x_2 - x &= x_2 - [x_1 - By_1(1 - Cy_2)] \\ &= x_2 - x_1 + By_1(1 - Cy_2) \end{aligned}$$

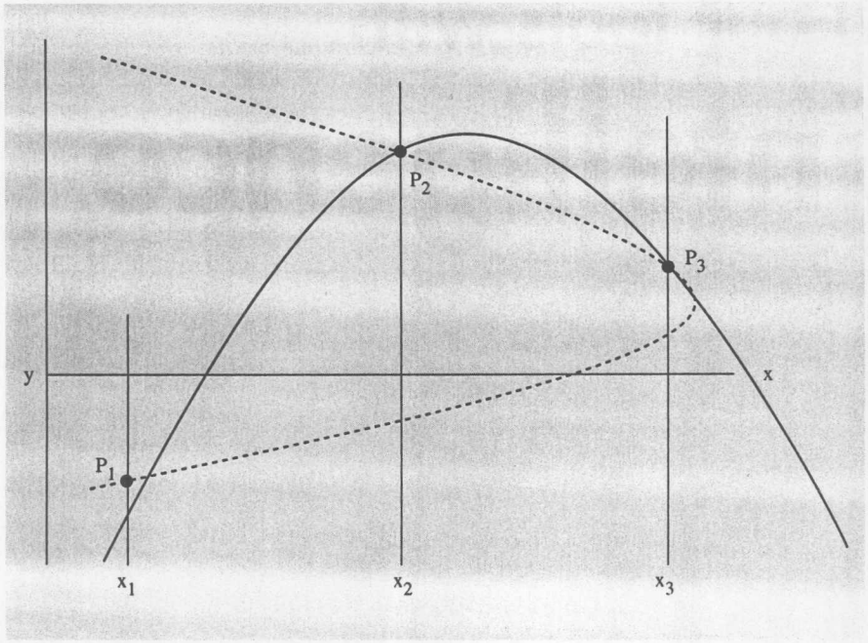
or:

$$x_2 - x = h + By_1(1 - Cy_2) \qquad (16)$$

(Because of our relabeling, h may no longer be positive.)

Substituting for B from Equation 7, we get:

FIGURE 4
A poor fit.



$$x_2 - x = h + \frac{hy_1(1 - Cy_2)}{y_2 - y_1}$$

$$= h\left[1 + \frac{y_1(1 - Cy_2)}{y_2 - y_1}\right]$$

$$= h\left[\frac{y_2 - y_1 + y_1(1 - Cy_2)}{y_2 - y_1}\right]$$

or:

$$x_2 - x = hy_2\left(\frac{1 - Cy_1}{y_2 - y_1}\right) \tag{17}$$

Similarly:

$$x_1 - x = hy_1\left(\frac{1 - Cy_2}{y_2 - y_1}\right) \tag{18}$$

Then the inequality in Equation 15 becomes:

$$-h^2 y_1 y_2\left[\frac{(1 - Cy_1)(1 - Cy_2)}{(y_2 - y_1)^2}\right] \geq 0$$

Now, h^2 and the denominator are, of course, perfect squares, so their signs won't affect the outcome. Also, we know from Equation 13 that the sign of y_1y_2 must be negative. This leaves us with the condition:

$$(1 - Cy_1)(1 - Cy_2) \geq 0 \tag{19}$$

Next, we must substitute the value for C from Equation 11. After some truly horrible algebra that I'll spare you, we find:

$$1 - Cy_1 = \frac{y_3(y_3 - y_2) - y_1(y_2 - y_1)}{(y_3 - y_2)(y_3 - y_1)} \tag{20}$$

And:

$$1 - Cy_2 = \frac{y_3(y_3 - y_1) - 2y_2(y_2 - y_1)}{(y_3 - y_2)(y_3 - y_1)} \tag{21}$$

Let:

$$u = y_3(y_3 - y_2) - y_1(y_2 - y_1)$$
$$v = y_3(y_3 - y_1) - 2y_2(y_2 - y_1) \tag{22}$$

Then we have:

$$1 - Cy_1 = \frac{u}{(y_3 - y_2)(y_3 - y_1)}$$
$$1 - Cy_2 = \frac{v}{(y_3 - y_2)(y_3 - y_1)} \tag{23}$$

When we multiply these equations together, we again will get perfect squares in the denominator, which won't affect the sign of the result. So our inequality becomes, finally:

$$uv \geq 0 \tag{24}$$

We can identify four cases, depending on the signs of u and v:

- Case 1: u > 0, v > 0 OK
- Case 2: u > 0, v < 0 Forbidden
- Case 3: u < 0, v > 0 Forbidden
- Case 4: u < 0, v < 0 OK

PUSHING THE ENVELOPE

Now, we must somehow relate these cases to the corresponding values of the function at points P_1, P_2, and P_3. We can explore the boundaries between the cases by setting u and v, separately, equal to zero. For example, when u = 0, we have:

$$y_3(y_3 - y_2) - y_1(y_2 - y_1) = 0 \tag{25}$$

This equation is quadratic in y_1 and y_3, but only linear in y_2, so we can solve for that parameter. We get:

$$y_2(y_1 + y_3) = y_1^2 + y_3^2$$

or:

$$y_2 = \frac{y_1^2 + y_3^2}{y_1 + y_3} \tag{26}$$

At this point, let's introduce two new variables:

$$\beta = \frac{y_2}{y_1};\ \gamma = \frac{y_3}{y_1} \tag{27}$$

Because of our restrictions on the ranges (see Equations 13 and 14), both β and γ must be negative.

Using these definitions, we can rewrite Equation 26 in the form:

$$\beta = \frac{1 + \gamma^2}{1 + \gamma} \tag{28}$$

Rearranging the terms gives:

$$\gamma^2 - \beta\gamma + 1 - \beta = 0 \tag{29}$$

This is a quadratic in γ, which we can

solve using the quadratic formula. Let:

$$a = 1;\ b = -\beta;\ c = 1 - \beta \qquad (30)$$

Then the quadratic formula gives:

$$\gamma = \frac{\beta \pm \sqrt{\beta^2 - 4(1-\beta)}}{2} \qquad (31)$$

This curve is plotted as the lighter curve in Figure 5. As you can see, it has two lobes, enclosing the two areas marked "Case 4."

We can get the other boundary by setting v = 0. From Equation 22, this gives:

$$y_3(y_3 - y_1) - 2y_2(y_2 - y_1) = 0 \qquad (32)$$

This time, it's y_1 that we can solve for:

$$y_1 = \frac{y_3^2 - 2y_2^2}{y_3 - 2y_2} \qquad (33)$$

Using Equation 27 as before, this becomes:

$$1 = \frac{\gamma^2 - 2\beta^2}{\gamma - 2\beta}$$

A bit of rearranging gives us:

$$\gamma - 2\beta = \gamma^2 - 2\beta^2$$

or:

$$2\beta^2 - \gamma^2 - 2\beta + \gamma = 0 \qquad (34)$$

This equation is a little different from the previous one because it's a quadratic in both β and γ. In fact, it represents a hyperbola. With a little algebra that's too long to show here, we can prove that the hyperbola opens along the γ-axis, is centered at the point (1/2, 1/2), and has limiting values of β given by:

$$\beta = \tfrac{1}{2}\left(1 \pm \tfrac{1}{4}\sqrt{2}\right) \qquad (35)$$

I'm not showing the precise form of the equation for the hyperbola because we don't really need it. We can simply solve Equation 34 for γ, just as we did for Equation 29. The equation has the form of a quadratic with:

$$a = -1;\ b = 1;\ c = 2\beta\,(\beta - 1) \qquad (36)$$

The quadratic formula gives us:

$$\gamma = \tfrac{1}{2}\left(1 \pm \sqrt{1 - 8\beta\,(\beta - 1)}\right) \qquad (37)$$

This curve is plotted as the heavier line in Figure 5. Between the two curves, defined by u = 0 and v = 0, we've defined the boundaries of all four cases. You might be wondering what happened to Case 3, since it's not shown on the curve. As it turns out, this case is represented by a tiny sliver where the two curves intersect near the point $\beta = 0.85$, $\gamma = 0.5$. The region is too small to show up on this graph. Its precise location and shape turn out to

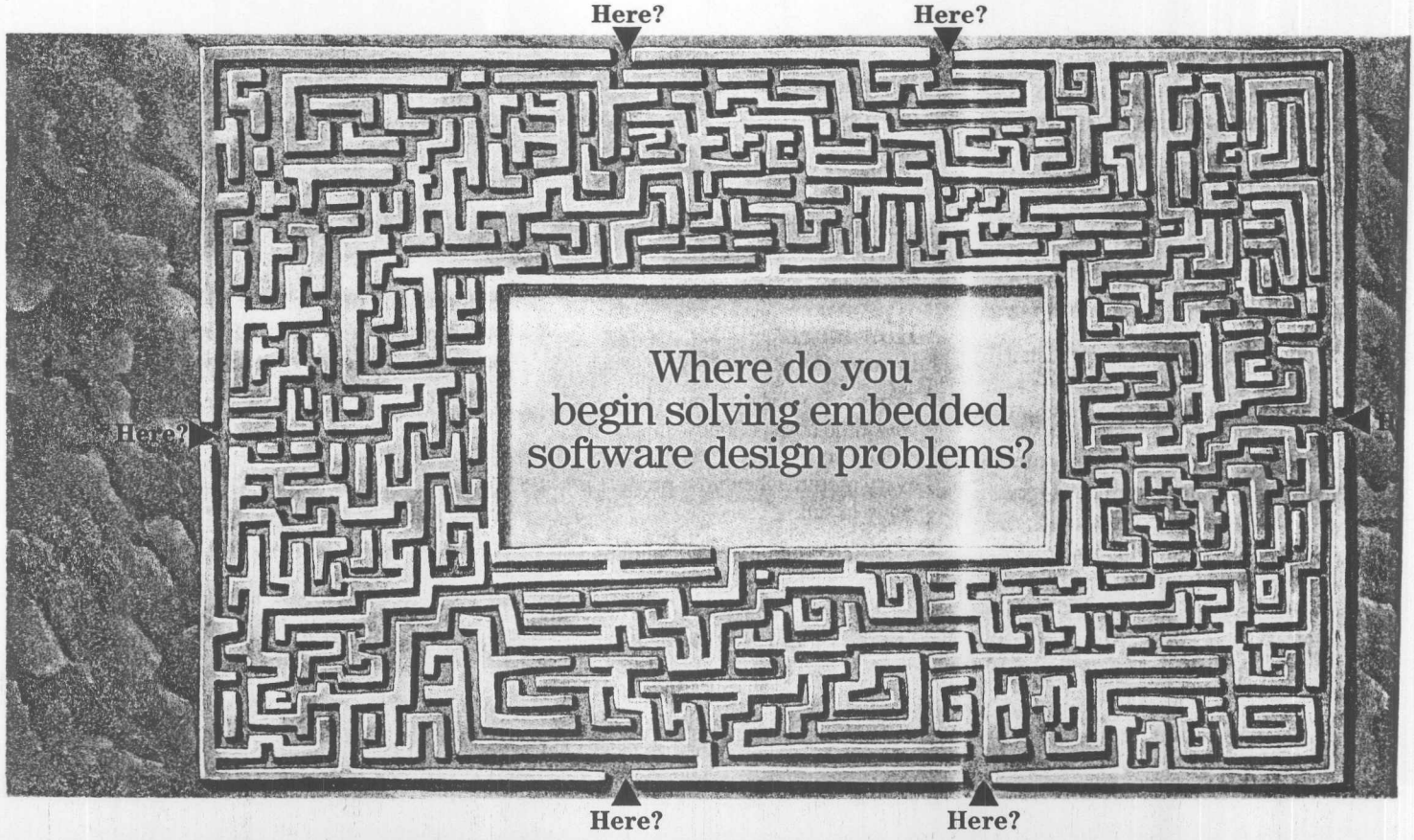
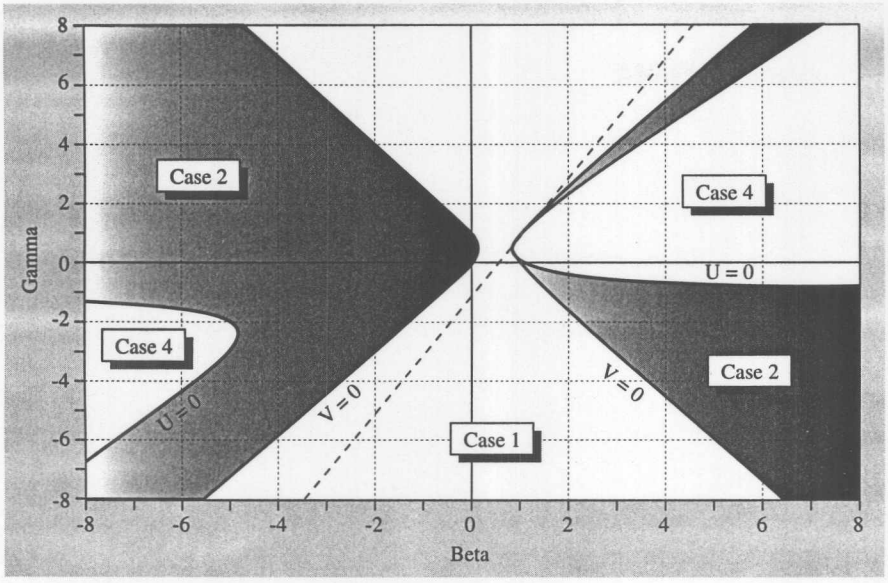


FIGURE 5
Boundaries.



be immaterial anyway because, while we've shown the graph for positive and negative values of β and γ, the restrictions given by Equations 13 and 14 demand that β and γ be negative. Thus, we're really only concerned with the lower left-hand quadrant of Figure 5.

WHICH REGION?

Only Cases 1 and 4 satisfy the inequality in Equation 24. Within that lower left quadrant of Figure 5, we are allowed to operate in the triangular-shaped part of Case 1 and the part of Case 4 curving in from the left.

I'm going to assert now that only the Case 1 portion will do us any good. Here's why: Consider what happens as we start getting closer to convergence. Presumably, our boundary points P_1 through P_3 will get closer and closer together, and the segment of the function they define will become more nearly a straight line. When the curve is a straight line, the quadratic of Equation 5 degenerates as the quadratic coefficient C approaches zero. Let's see what must happen to β and γ as this degeneration takes place. Setting C to zero in Equation 11 gives us:

$$-y_1 + 2y_2 - y_3 = 0$$

or the following:

$$2\beta - \gamma = 1 \qquad (38)$$

This, of course, is a straight line, which I've plotted as the dashed line in Figure 5. As we are iterating and getting closer to the true root, we can expect the values for each iteration to migrate toward this line; the actual point will depend on the slope of the function curve. In the lower-left quadrant, this line lies wholly in the Case 1 region. Thus our point can indeed migrate to the straight line. The Case 4 region, on the other hand, is cut off from the solution line by the Case 2 region surrounding it. Because of this, starting a quadratic step from within Case 4 is a fruitless exercise; we'll end up having to switch back to bisection anyway. Case 4, in fact, corresponds to the case of an acceptable, but poor, fit similar to that of Figure 4.

What we've gained out of all these equations and graphs, then, is the conviction that only the small triangular region in the lower right corner of the lower left quadrant will be suitable for inverse quadratic interpolation. In all other regions, we should stick with bisection.

Armed with this conviction, we're finally prepared to define a useful criterion for accepting the interpolated root. Look again at the four cases; the only difference between Case 1 and Case 3 is the sign of u. If we had to worry about falling into Case 3, we would have to test the signs of both u and v. Since Case 3 only lies in that tiny sliver in the upper right quadrant, we needn't worry about it. We can never fall into that region. We can ignore the sign of u and accept as our criterion the condition:

$$v>0 \qquad (39)$$

The value of v is given by Equation 22. This leads us to a simple test for acceptable interpolation:

$$y_3(y_3 - y_1) - 2y_2(y_2 - y_1) > 0 \qquad (40)$$

Having struggled with all those equations and graphs, you can now forget them. They served only to establish the condition in Equation 40.

THE ALGORITHM

Now we can give the complete algorithm for Method X:

- Given two points x_1 and x_3 straddling a root, bisect the region:

$$x_2 = \frac{x_1 + x_3}{2} \qquad (41)$$

and evaluate the function to get y_2.
- Test the sign of y_2y_3. If it is negative, exchange the points P_1 and P_3 to make it positive.
- Test the sign of:

$$v = y_3(y_3 - y_1) - 2y_2(y_2 - y_1)$$

If the sign is negative, replace P_3 by P_2 and repeat. If the sign is positive, compute the coefficients:

$$B = \frac{x_2 - x_1}{y_2 - y_1}$$

and:

$$C = \frac{-y_1 + 2y_2 - y_3}{(y_3 - y_2)(y_3 - y_1)}$$

- Compute the estimated root from:

$$x = x_1 - By_1(1 - Cy_2)$$

and evaluate the new function value, y = f(x).
- Test the sign of y to determine which points to use for the next cycle. If $yy_1 < 0$, replace P_3 by (x,y). Otherwise replace P_1 by (x,y) and P_3 by P_2.
- Repeat until two successive roots are equal within a specified error, epsilon.

IMPLEMENTATION

Now that we finally have the algorithm, implementing it in code is straightforward enough. The results are shown in Listing 1. I tested the algorithm using our now-standard test function, which is shown on the following page:

LISTING 1
Second-order iteration.

```
enum {none, bad_data, no_convergence}
error;

// iterative root-finder using
// alternating bisection and inverse
// quadratic interpolation

// call  as: x2 = iterate(x1,x3,f,
// eps,imax,error);
// where x1,x3: initial points
// bracketing root
// f(x): name of user-supplied function
// eps: accuracy required in x
// imax: maximum number of iterations
// function return value contains result
// error: output flag
// error=none -> normal return
// error=bad_data -> initial points
// don't bracket root
// error=no_convergence -> cannot
// converge in imax tries

double iterate(double x1,double x3,
    double (*f)(double),double eps,int
      imax,int & error){
  double x2,y1,y2,y3,b,c,temp,y21,y31,
    y32,xm,ym;
     error=none;
  y1=f(x1);
  if(y1==0.0)return x1;
  y3=f(x3);
  if(y3==0.0)return x3;
  if(y3*y1>0.0){
    error=bad_data;
    return x1;
  }
  for(int i=0;i<imax;i++){
    x2=0.5*(x3+x1);
    y2=f(x2);
    if(y2==0.0)return x2;
    if(abs(x2-x1)<eps)return x2;
    if(y2*y1>0.0){
      temp=x1;
      x1=x3;
      x3=temp;
      temp=y1;
      y1=y3;
     y3=temp;
    }
    y21=y2-y1;
    y32=y3-y2;
    y31=y3-y1;
    if(y3*y31<2.0*y2*y21){
      x3=x2;
```

```
        y3=y2;
    }
    else{
      b=(x2-x1)/y21;
      c=(y21-y32)/(y32*y31);
      xm=x1-b*y1*(1.0-c*y2);
      ym=f(xm);
      if(ym==0.0)return xm;
      if(abs(xm-x1)<eps)return xm;
      if(ym*y1<0.0){
        x3=xm;
        y3=ym;
        }
      else{
        x1=xm;
        y1=ym;
        x3=x2;
        y3=y2;
       }
    }
  }
  error=no_convergence;
  return x2;
}
```

TABLE 1
Iterative results.

Iterations	Estimated Root	Error
1	4.917669	1.515485
2	3.214061	0.188123
3	3.398273	0.003911
4	3.402173	0.000011
5	3.402184	0

$$f(x) = x - 4\sin x + e^{-x/6} - 5$$

The results are shown in Table 1. As you can see, the speed with which the algorithm locks in on the root, once it's gotten in the neighborhood, is impressive indeed. William Press writes that Brent's method is the method of choice for general-purpose use. Although either Brent's method or Method X should give equally good performance, since they use the same philosophies, I claim that Method X is the method of choice because the equations are a tad simpler. Use this function in good health. You won't find a better one anywhere in the known universe. ESP

Jack W. Crenshaw is a staff scientist at In Vivo Research in Orlando, FL. He has developed numerous analysis and real-time programs and holds a PhD in physics from Auburn University. Crenshaw can be reached via e-mail at 72325.1327@compuserve.com.

REFERENCES

1. Press, William H., et al, *Numerical Recipes in C*, New York, NY: Cambridge University Press, 1988, pp. 267-269.